Supplementary Materials

Srinivasan Iyer[†], Alvin Cheung[§] and Luke Zettlemoyer[†‡]
[†]Paul G. Allen School of Computer Science and Engineering, Univ. of Washington, Seattle, WA
{sviyer, lsz}@cs.washington.edu
§Department of Electrical Engineering and Computer Sciences, UC Berkeley, Berkeley, CA
akcheung@cs.berkeley.edu
‡ Facebook AI Research, Seattle
lsz@fb.com

1 Iyer-Simp for CONCODE

Iyer-Simp is similar to the best performing encoder-decoder model of Iyer et al. (2018) on the CONCODE dataset, with three major modifications in their encoder, which yields improvements in speed and accuracy. First, in addition to camel-case splitting of identifier tokens, we use byte-pair encoding (BPE) (Sennrich et al., 2016) on all NL tokens, identifier names and types and embed these BPE tokens using a single embedding matrix. Next, we replace their RNN that contextualizes the subtokens of identifiers and types with an average of the subtoken embeddings instead. Finally, we consolidate their three separate RNNs for contextualizing NL, variable names with types, and method names with types, into a single shared RNN, which greatly cuts down model parameters. We present the full model here for convenience. Since the decoder is unmodified, large portions of this section are borrowed from Iyer et al. (2018).

Formally, let $\{q_i\}$ represent the set of BPE tokens of the NL, and $\{t_{ij}\}$, $\{v_{ij}\}$, $\{r_{ij}\}$ and $\{m_{ij}\}$ represent the jth BPE token of the ith variable type, variable name, method return type, and method name respectively.

Encoder The encoder computes contextual representations of the NL and each component in the context. First, all the elements defined above are embedded using a BPE token embedding matrix B to give us $\mathbf{q_i}$, $\mathbf{t_{ij}}$, $\mathbf{v_{ij}}$, $\mathbf{r_{ij}}$ and $\mathbf{m_{ij}}$. Using Bi-LSTM f, the encoder then computes:

$$h_1, \cdots, h_z = f(\mathbf{q_1}, \ldots, \mathbf{q_z}) \quad (1)$$

$$\mathbf{v_i} = Avg(\mathbf{v_{i1}}, \ldots, \mathbf{v_{ij}}) \quad (2)$$

$$\text{Similarly, compute } \mathbf{m_i}, \mathbf{t_i}, \mathbf{r_i} \quad (3)$$

$$\hat{t_i}, \hat{v_i} = f(\mathbf{t_i}, \mathbf{v_i}) \quad (4)$$

$$\hat{r_i}, \hat{m_i} = f(\mathbf{r_i}, \mathbf{m_i}) \quad (5)$$

Then, $h_1, \ldots, h_z$, and $\hat{t_i}, \hat{v_i}, \hat{r_i}, \hat{m_i}$ are passed to the attention mechanism in the decoder.

Decoder The decoder is a sequential LSTM based model that produces a sequence of grammar rules (a_t at step t), which can later be put together to form a source code snippet. At each time step t, the decoder expands a non-terminal that was produced earlier, by choosing a valid right hand side rule for that non-terminal. The first non-terminal (at step 1) is *MemberDeclaration* and subsequently, every non-terminal is expanded in a depth first left to right fashion, similar to Yin and Neubig (2017). The probability of a source code snippet is decomposed as a product of the conditional probability of generating each step in the sequence of rules conditioned on the previously generated rules.

More specifically, the decoder is an LSTM-based RNN with hidden dimension size H, that produces a context vector c_t at each time step, which is used to compute a distribution over next actions.

$$p(a_t|a_{<t}) \propto exp(W^{n_t} c_t) \quad (6)$$

Here, W^{n_t} is a $|n_t| \times H$ matrix, where $|n_t|$ is the total number of unique grammar rules that n_t can be expanded to. The context vector c_t is computed using the hidden state s_t of an n-layer decoder LSTM cell and attention vectors over the NL and the context (z_t and e_t), as described below.

Decoder LSTM The decoder uses an n-layer LSTM whose hidden state s_t is computed based on the current non-terminal n_t to be expanded, the previous production rule a_{t-1}, the parent production rule, par(n_t), that produced n_t, the previous decoder LSTM state s_{t-1}, and the decoder state of the LSTM cell that produced n_t, denoted as s_{n_t}.

$$s_t = \text{LSTM}(n_t, a_{t-1}, \text{par}(n_t), s_{t-1}, s_{n_t}) \quad (7)$$

We use an embedding matrix N to embed n_t and matrix A to embed a_{t-1} and par(n_t). If a_{t-1} is a rule that generates a terminal node that represents an identifier or literal, it is represented using a special rule *IdentifierOrLiteral* to collapse all these rules into a single previous rule.

Two-step Attention At time step t, the decoder first attends to every token in the NL representation, h_i, using the current decoder state, s_t, to compute a set of attention weights α_t, which are used to combine h_i into an NL context vector z_t. We use a general attention mechanism (Luong et al., 2015), extended to perform multiple steps.

$$\alpha_{t,i} = \frac{\exp(s_t^{\mathrm{T}} \mathbf{F} h_i)}{\sum_i \exp(s_t^{\mathrm{T}} \mathbf{F} h_i)}$$
$$z_t = \sum_i \alpha_{t,i} h_i$$

The context vector z_t is used to attend over every type (return type) and variable (method) name in the environment, to produce attention weights β_t that are used to combine the entire context $x = [t : v : r : m]$ into an environment context vector e_t.[1]

$$\beta_{t,j} = \frac{\exp(z_t^{\mathrm{T}} \mathbf{G} x_j)}{\sum_j \exp(z_t^{\mathrm{T}} \mathbf{G} x_j)}$$
$$e_t = \sum_j \beta_{t,j} x_j$$

Finally, c_t is computed using the decoder state and both context vectors z_t and e_t:

$$c_t = tanh(\hat{W}[s_t : z_t : e_t])$$

Supervised Copy Mechanism Since the class environment at test time can belong to previously unseen new domains, our model needs to learn to copy variables and methods into the output. We use the copying technique of Gu et al. (2016) to compute a copy probability at every time step t using learned vector b of dimensionality H.

$$\text{copy}(t) = \sigma(b^T c_t)$$

Since we only require named identifiers or user defined types to be copied, both of which are produced by production rules with n_t as *IdentifierNT*, we make use of this copy mechanism only in this case. Identifiers can be generated by directly generating derivation rules (see equation 6), or by copying from the environment. The probability of copying an environment token x_j, is set to be the attention weights $\beta_{t,j}$ computed earlier, which attends exactly on the environment types and names which we wish to be able to copy. The copying process is supervised by preprocessing the grammar rules to recognize identifiers that can be copied from the environment, and both the generation and the copy probabilities are weighted by $1 - \text{copy}(t)$ and copy(t) respectively.

Hyperparameters and Inference We use an embedding size H of 1024 for identifiers and types. All LSTM cells use 2-layers and a hidden dimensionality of 1024 (512 on each direction for BiLSTMs). We use an embedding size of 512 for encoding non-terminals and rules in the decoder. We use dropout with $p = 0.5$ in between LSTM layers and at the output of the decoder over c_t. We train our model for 30 epochs using mini-batch gradient descent with a batch size of 40, and we use Adam (Kingma and Ba, 2015) with an initial learning rate of 0.001 for optimization. We decay our learning rate by 80% based on performance on the development set after every epoch. We use beam search with a beam size of 5 for decoding the sequence of grammar rules at test time.

2 Seq2Prod for ATIS-SQL

Our Seq2Prod is similar to the Seq2Prod model of Iyer et al. (2018), with a modification in the inputs to the decoder LSTM. We describe the entire model here for convenience. Large portions of this section are borrowed from Iyer et al. (2018).

Encoder The encoder of Seq2Prod computes contextual representations of the NL query. Note that unlike the previous model for CONCODE, this model does not need to encode context. If q_i represents each lemmatized token of the NL, they are first embedded using a token embedding matrix B to give us $\mathbf{q_i}$. Using Bi-LSTM f, the encoder then computes:

$$h_1, \cdots, h_z = f(\mathbf{q_1}, \ldots, \mathbf{q_z}) \quad (8)$$

Then, $h_1, \ldots, h_z$ are passed to the attention mechanism in the decoder.

Decoder Similar to the Iyer-Simp model described above, the decoder is a sequential LSTM based model that produces a sequence of grammar rules (a_t at step t), which can later be put together

[1] ":" denotes concatenation.

to form a source code snippet. More specifically, the decoder is an LSTM-based RNN with hidden dimension size H, that produces a context vector c_t at each time step, which is used to compute a distribution over next actions.

$$p(a_t|a_{<t}) \propto exp(W^{n_t} c_t) \quad (9)$$

Here, W^{n_t} is a $|n_t| \times H$ matrix, where $|n_t|$ is the total number of unique grammar rules that n_t can be expanded to. The context vector c_t is computed using the hidden state s_t of an n-layer decoder LSTM cell and attention vectors over the NL z_t, as described below.

Decoder LSTM The decoder uses an n-layer LSTM (LSTM_f) whose hidden state at time t, s_t, is computed based on an embedding of the current non-terminal n_t to be expanded, a contextuatized embedding of the previous production rule a_{t-1}, a contextualized embedding of the parent production rule, $\text{par}(n_t)$, that produced n_t, and the previous decoder LSTM state s_{t-1}. Note that unlike the previous Iyer-Simp model, we do not use the parent LSTM state as it does not provide any improved performance.

We use an embedding matrix N to embed n_t. To compute the contextualized embeddings of a_{t-1} and $\text{par}(n_t)$ in LSTM_f, we use another single layer Bi-LSTM (LSTM_g) across the left and right sides of the rule (using separator symbol SEP) and use the final hidden state as inputs to LSTM_f instead. More concretely, if a grammar rule is represented as $A \rightarrow B_1 \ldots B_n$, then:

$$\text{Emb}(A \rightarrow B_1 \ldots B_n) = \text{LSTM}_g(\{A, \texttt{SEP}, B_1, \ldots, B_n\}) \quad (10)$$

$$s_t = \text{LSTM}_f(n_t, \text{Emb}(a_{t-1}), \text{Emb}(\text{par}(n_t)), s_{t-1}) \quad (11)$$

The contextualization of these rule embeddings is the primary difference between our model and Iyer et al. (2018). This modification can help the LSTM_f cell locate the position of n_t within rules a_{t-1} and $\text{par}(n_t)$, especially, for lengthy idiomatic rules.

s_t is then used for attention and finally, produces a distribution over grammar rules.

Single-Step Attention At time step t, the decoder attends to every token in the NL representation, h_i, using the current decoder state, s_t, to compute a set of attention weights α_t, which are used to combine h_i into an NL context vector z_t. We use the general attention mechanism of Luong et al. (2015).

$$\alpha_{t,i} = \frac{\exp(s_t^{\mathrm{T}} \mathbf{F} h_i)}{\sum_i \exp(s_t^{\mathrm{T}} \mathbf{F} h_i)}$$
$$z_t = \sum_i \alpha_{t,i} h_i$$

Finally, c_t is computed using the decoder state and context vector z_t:

$$c_t = tanh(\hat{W}[s_t : z_t])$$

Supervised Copy Mechanism The supervised copy mechanism is exactly the same as described for the previous model (Iyer-Simp), using context vector c_t.

Hyperparameters We use an embedding size H of 1024 for NL query tokens. Both the encoder and decoder LSTM cells use 2-layers and a hidden dimensionality of 1024 (512 on each direction for BiLSTMs). We use an embedding size of 512 for encoding non-terminals and a hidden size of 256 for the contextualized rules (for LSTM_g) in the decoder. We use dropout with $p = 0.5$ in between LSTM layers and at the output of the decoder over c_t. We train our model for 60 epochs using mini-batch gradient descent with a batch size of 40, and we use Adam (Kingma and Ba, 2015) with an initial learning rate of 0.001 for optimization. We decay our learning rate by 80% based on performance on the development set after every epoch. We use beam search with a beam size of 5 for decoding the sequence of grammar rules at test time.

References

Jiatao Gu, Zhengdong Lu, Hang Li, and Victor O.K. Li. 2016. Incorporating copying mechanism in sequence-to-sequence learning. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 1631–1640, Berlin, Germany. Association for Computational Linguistics.

Srinivasan Iyer, Ioannis Konstas, Alvin Cheung, and Luke Zettlemoyer. 2018. Mapping language to code in programmatic context. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*, pages 1643–1652. Association for Computational Linguistics.

Diederik Kingma and Jimmy Ba. 2015. Adam: A method for stochastic optimization. In *ICLR*.

Thang Luong, Hieu Pham, and D. Christopher Manning. 2015. Effective approaches to attention-based neural machine translation. In *Proceedings of the 2015 Conference on Empirical Methods in Natural Language Processing*, pages 1412–1421. Association for Computational Linguistics.

Rico Sennrich, Barry Haddow, and Alexandra Birch. 2016. Neural machine translation of rare words with subword units. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 1715–1725, Berlin, Germany. Association for Computational Linguistics.

Pengcheng Yin and Graham Neubig. 2017. A syntactic neural model for general-purpose code generation. In *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 440–450, Vancouver, Canada. Association for Computational Linguistics.